

Received January 7, 2020, accepted February 4, 2020, date of publication February 13, 2020, date of current version February 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2973750

Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study

YIWEN WU^{ID}, YANG ZHANG^{ID}, TAO WANG^{ID}, AND HUAIMIN WANG^{ID}

Science and Technology on Parallel and Distributed Laboratory, National University of Defense Technology, Changsha 410073, China
Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, Changsha 410073, China

Corresponding author: Yang Zhang (yangzhang15@nudt.edu.cn)

This work was supported in part by the Program of a New Generation of Artificial Intelligence 2030 under Grant 2018AAA0102304, in part by the Foundation of PDL under Grant 6142110190204, and in part by the National Natural Science Foundation of China under Grant 61702532.

ABSTRACT Dockerfile plays an important role in the Docker-based software development process, but many Dockerfile codes are infected with smells in practice. Understanding the occurrence of Dockerfile smells in open-source software can benefit the practice of Dockerfile and enhance project maintenance. In this paper, we perform an empirical study on a large dataset of 6,334 projects to help developers gain some insights into the occurrence of Dockerfile smells, including its coverage, distribution, co-occurrence, and correlation with project characteristics. Our results show that smells are very common in Dockerfile codes and there exists co-occurrence between different types of Dockerfile smells. Further, using linear regression analysis, when controlled for various variables, we statistically identify and quantify the relationships between Dockerfile smells occurrence and project characteristics. We also provide a rich resource of implications for software practitioners.

INDEX TERMS Docker, Dockerfile smells, Open-source software, GitHub.

I. INTRODUCTION

“There are over one million Dockerfiles on GitHub today, but not all Dockerfiles are created equally.” — Tibor Vass¹

Docker², as one of the most popular containerization tools, enables the encapsulation of software packages into containers [1]. Docker allows packaging an application with its dependencies and execution environment into a standardized, self-contained unit, which can be used for software development and to run the application on any system [2]. Since inception in 2013, Docker containers have gained 32,000+ GitHub stars and have been downloaded 105B+ times³. The “Annual Container Adoption” report⁴ found that 79% of companies chose Docker as their primary container technology. The contents of a Docker container are defined by

declarations in the **Dockerfile** [3] which specifies the Docker commands and the order of their execution, following the notion of Infrastructure-as-Code (IaC) [4]. Thus, studying Dockerfile is very relevant to Docker-based software development.

Code smells [5] indicate the presence of quality problems in a software project. Recently, smell metaphor has been extended to various related sub-domains of software, e.g., database [6], logging [7], and continuous integration [8]. Typically, when developers are building a Docker image, they should thoroughly read Docker’s official documentation’s best practices for Dockerfile⁵. Although such guideline covers the recommended best practices and methods, it is still challenging for developers to fully follow the recommended rules due to lack of awareness and attention. Therefore, similar to regular code, Dockerfile code can also indicate smells. However, the presence/absence of Dockerfile smells in OSS projects and their relationships with project characteristics

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone^{ID}.

¹<https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/>

²<https://www.docker.com/>

³<https://www.docker.com/company, as of November 2019>

⁴<https://portworx.com/2017-container-adoption-survey/>

⁵https://docs.docker.com/develop/develop-images/dockerfile_best-practices

have not been extensively explored yet. Better understanding the occurrence of Dockerfile smells can shed light on the current compliance and correctness of Dockerfile configuration in OSS projects, thereby fostering the awareness and attention of developers and informing the development of enhanced Dockerfile configuration guidelines.

In this paper, we present an empirical study of the occurrence of Dockerfile smells in a large-scale dataset of GitHub projects, to help developers gain insights into the Dockerfile smells, including its coverage, distribution, co-occurrence, and the correlation with project characteristics. In our study, Dockerfile smells are divided into two specific smells, i.e., DL-smells (violate the official Dockerfile best practices) and SC-smells (violate the basic shell scripts practices), the detailed description can be found in Section II-A.2. During our analysis, we collect and analyze the data from more than 6,000 GitHub open-source projects. First, we quantitatively investigate the basic smells coverage and its difference between different types of projects, i.e., different owner types and programming languages. Next, we investigate the distribution of DL-smells and SC-smells, and discuss their co-occurrence. Moreover, to understand whether Dockerfile smells occurrence depends on the characteristics of projects, we develop three linear regression models, by controlling for various confounds. Finally, we distill many practical implications for researchers, developers, and tool builders.

The highlights of our findings are:

- Nearly 84% of GitHub projects in our dataset have smells in their Dockerfile code, especially the DL-smells. On average, 62% of Dockerfile instructions are infected with smells.
- DL-smells appear more than SC-smells. Moreover, a project that shows presence of large number of DL-smells (or SC-smells) moderately indicates presence of large number of SC-smells (or DL-smells).
- Dockerfile smells coverage, distribution, co-occurrence in projects with different programming languages are different too.
- The occurrence of Dockerfile smells is correlated with some project characteristics. Popular and young projects, or projects with large contributors team, tend to have fewer Dockerfile smells.

To the best of our knowledge, this is a first empirical study of characterizing Dockerfile smells occurrence in Open-Source software. Our dataset can be found online at IEEE DataPort⁶. The remaining of this paper is structured as follows. In Section II, we present the preliminaries. In Section III, we discuss the methods used to collect and analyze data. In Section IV, we present empirical results for our research question. In Section V, we discuss qualitative analysis, practical implications and the limits of our study. We finally conclude this paper in Section VI.

⁶<http://dx.doi.org/10.21227/r9v8-4f07>

Base Image	← FROM debian DL Smell: Always tag the version of an image explicitly
Maintainer Info	← MAINTAINER David James<david@james.com> DL Smell: MAINTAINER is deprecated
ENV. Variable	← ENV USE_HTTP 1
Comment	← # Add proxy settings
Copy script	← COPY ./setenv.sh /tmp/
Execute shell	← RUN export node_version="0.10"
Installing dependencies	{ RUN apt-get update RUN apt-get -y install nodejs="\$node_version" SC Smell: node_version is referenced but not assigned (did you mean 'node_version')? DL Smell: Avoid additional packages by specifying --no-install-recommends
Copy file	← COPY package.json usr/src/app
Installing software	{ RUN cd /usr/src/app SC Smell: Use 'cd ... exit' or 'cd ... return' in case cd fails. DL Smell: Use WORKDIR to switch to a directory RUN npm install node-static DL Smell: Pin versions in npm. Instead of 'npm install <package>' use 'npm install <package>@<version>'
Open Port	← EXPOSE 80000 DL Smell: Valid UNIX ports range from 0 to 65535
Start Process	← CMD ["npm", "start"]

FIGURE 1. Example of a Dockerfile illustrating instructions and smells.

II. PRELIMINARIES

A. DOCKERFILE AND DOCKERFILE SMELLS

1) DOCKERFILE

In Docker, Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image [9]. Docker runs instructions in a Dockerfile and all instructions must start with a base image. Other parts are then added on top of the base one [1]. Docker has provided multiple types of instructions in the Dockerfile, involving FROM, RUN, COPY, ADD, ENV, CMD, EXPOSE, etc. Figure 1 shows an example of Dockerfile, which has 12 instructions and 1 comment. Specifically, the FROM instruction specifies the base image, which can give a first indication of what it is that the projects use Docker for [10], e.g., “debian”. The MAINTAINER instruction provides the name and email of an active maintainer. And the ENV instruction sets the environment variables. The COPY instruction places files into the container, and the RUN instruction executes any possible shell commands in a new layer on top of the current image and commits the results. The EXPOSE informs Docker that the container listens on the specified network ports at runtime. And the CMD provides defaults for an executing container.

2) DOCKERFILE SMELLS

As we know, Dockerfile has a specific format and uses a specific set of instructions. Similar to traditional production code, Dockerfile code may also become unmaintainable if the configuration instructions of Dockerfile are made without diligence and care. Some researches [11] argued that configuration code must be treated as production code due to the characteristics and maintenance needs of the configuration

code. Thus, best Dockerfile practices must be adopted to write and maintain high quality Dockerfile code. Due to limitations of experience and capabilities, developer's Dockerfile code may violate the recommended best practices, thereby causing Dockerfile smells.

Similar to the traditional configuration code smell [12], **Dockerfile smells** are the instructions of a Dockerfile that violate the recommended best practices and potentially affect the Dockerfile's quality in a negative way. Thus, in our study, if a project's Dockerfile has at least one smell, we call this project as **Smell project** or **S-project**, otherwise, this project is a **Healthy project** or **H-project**. Further, Dockerfile smells can be divided into two major categories: DL-smells and SC-smells.

DL-smells refer to the issues against the rules of Dockerfile best practices. Those smells can be typical design issues that cause a Dockerfile to fail when building images, or typical implementation issues that cause the building latency of a Dockerfile too long, even some architectural issues that may bring a Dockerfile the problem of maintainability, security, and reproducibility. E.g., in Figure 1, when developers defining the base image, they should always tag the version of an image explicitly. Also, developers should always use the valid UNIX ports (range from 0 to 65535) to open port, otherwise, it will violate the DL-smell.

SC-smells are quality issues against the basic rules of shell scripts practices. Those smells can be typical beginner's syntax issues that cause a shell to give cryptic error messages, or typical intermediate level semantic problems that cause a shell to behave strangely and counter-intuitively, as well as subtle caveats, corner cases, and pitfalls that may cause an advanced user's otherwise working script to fail under future circumstances. E.g., in the example of Figure 1, before installing the software *node-static*, Dockerfile runs the shell command "`cd /usr/src/app`", which violates the SC-smell, i.e., it should use '`cd ... || exit`' or '`cd ... || return`' in case *cd* fails.

B. RELATED WORK

1) STUDIES ON INFRASTRUCTURE AS CODE

Infrastructure as Code (IaC) [13], the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods, has attracted many researchers to investigate. Specifically, Xu *et al.* [14] studied the over-design issues in configurations and proposed some techniques to simplify the design space of configurations. They also analyzed issues related to security-related configurations to understand the reasons for these security misconfigurations [15]. Zhi *et al.* [7] conducted an exploratory study on logging configuration practice of 10 open-source projects and 10 industrial projects written in Java in various sizes and domains. They categorized and analyzed the change history of logging configurations to understand how the logging configurations evolve.

Sharma *et al.* [12] analyzed 4,621 Puppet repositories and detected the cataloged implementation and design configuration smells. Their analysis revealed that the design configuration smells showed 9% higher average co-occurrence among themselves than the implementation configuration smells.

Additionally, there exist studies to explore the practice of specific types of configurations, such as database schema configurations [6] and continuous integration configurations [8], [16]. Specifically, Sharma *et al.* [6] analyzed 357 industrial and 2,568 open-source projects and empirically studied quality characteristics of their database schemas. They found that the index abuse smell occurred most frequently in database code, and some database smells were more prone to occur in industrial projects compared to open-source projects. Gallaba and McIntosh [8] presented a study of feature use and misuse in 9,312 open-source systems that use Travis CI. Their results revealed that explicit deployment code was rare, however, 48.16% of the studied Travis CI specification code was instead associated with configuring job processing nodes. Vassallo *et al.* [16] verified that automated detection could help with early identification and prevented such a process decay. They conducted an empirical study on 18,474 CI build logs of 36 popular JAVA projects, and found that the presence of 3,823 high-severity warnings spread across projects.

Different from previous studies, our work here aims to understand the occurrence of smells in the Dockerfile configuration code. Moreover, despite the Dockerfile is widely used in GitHub projects, to the best of our knowledge, few research has investigated the detailed Dockerfile smells occurrence. With this paper, we attempt to address this literature gap, and provide insights into the occurrence of Dockerfile smells in the GitHub OSS community.

2) STUDIES ON DOCKER AND DOCKERFILE

Recently, many studies have been conducted on the Docker-based software development process. Specifically, Zhang *et al.* [17] proposed a mixed-methods study to shed light on developers' experiences and expectations with Docker-enabled workflows. Their results revealed two prominent workflows, based on the automated builds feature on Docker Hub or Continuous Integration services, with different trade-offs. Hassan *et al.* [18] proposed RUDSEA, a novel approach to recommend updates of Dockerfiles to developers based on analyzing changes on software environment assumptions and their impacts. Zhang *et al.* [19] studied Dockerfile longitudinal changes at a large scale and presented a clustering-based approach for mining Dockerfile evolutionary trajectories. They proposed an empirical study of 2,840 projects and found six distinct clusters of Dockerfile evolutionary trajectories. Further, they built two regression models to explore the impact of Dockerfile evolutionary trajectories and specific architecture attributes on Dockerfile quality and image build latency, which derived a number of suggestions for practitioners [20].

Also, some studies investigated the tag recommendation approach of Dockerfile and the Dockerfile Temporary File (TF) smells. Specifically, Yin *et al.* [21] used Labeled Latent Dirichlet Allocation (LDA) algorithm to recommend tags, by taking a Dockerfile as the specific text description. Recently, Zhou *et al.* [22] proposed a semi-supervised learning based tag recommendation approach, SemiTagRec, for Docker repositories, and their results obtained good performance. As for the Dockerfile TF smells, Lu *et al.* [23] made an empirical case study to the real-world Dockerfiles on Docker Hub. They summarized four different patterns of TF smells and proposed a state-depend static analysis method to detect them. Further, Xu *et al.* [24] proposed two different methods to detect TF smells in Dockerfile with dynamic analysis and static analysis respectively. Experimental results showed that their methods perform effectively.

While the existing literature helps researchers and software developers gain a deeper understanding of Dockerfile practices, few studies have investigated Dockerfile smells occurrence, the only exception being the study by Cito *et al.* [10]. In that paper, the authors reported on an exploratory study with the goal of characterizing the Docker ecosystem, prevalent quality issues, and the evolution of Dockerfile. Their study was based on sampling inspections of Top-100 and Top-1,000 most popular projects and found that most quality issues arise from missing version pinning in Dockerfile. Our work here is substantially different in two aspects. First, we consider a more refined set of studied projects (>6,000) than theirs, enabling powerful quantitative hypothesis testing and regression modeling, in addition to the basic statistics. Second, our goal is different, as we aim to comprehensively understand the coverage, distribution, co-occurrence of Dockerfile smells, as well as the correlation with project characteristics.

C. RESEARCH QUESTIONS

In this study, we aim to analyze the existing Dockerfile code and characterize the occurrence of Dockerfile smells, i.e., its coverage, distribution and co-occurrence, and correlation with project characteristics, to examine the best practices for writing Dockerfile. We formulate three research questions towards the characteristics of Dockerfile smells occurrence in GitHub OSS projects.

Dockerfile has been widely used in current Docker-based software development [10], [20]. Although Docker company provides official best practices, Dockerfiles in different projects may be created unequally. Analyzing the presence/absence of smells in Dockerfile codes would help us to better understand the current compliance and correctness of Dockerfile configuration in OSS projects. Thus, in our first question, we ask:

RQ1: How many projects have smells in their Dockerfile code? In this question, we seek to investigate the coverage of Dockerfile smells to find out how many projects violate

the rules of best practices and whether there exist some differences between different projects.

In addition to Docker instructions, developers can also write traditional Shell scripts in their Dockerfiles. Both of them may violate the best practices suggestions, thereby causing different types of Dockerfile smells. In traditional software engineering, it is said that “*no pattern is an island*”, i.e., if we find one smell, it is very likely that we will find many more around it [25]. Thus, investigating the co-occurrence of different types of Dockerfile smells can help us to find out whether the folklore is true in the context of Dockerfile smells. Furthermore, whether the co-occurrence in different types of projects follows the principle with the same degree. This would foster awareness and attention when developers writing Docker instructions and Shell scripts in the Dockerfile configuration. Therefore, in our second question, we ask:

RQ2: What is the distribution and relationship of different types of Dockerfile smells? In this question, we want to study the instances of DL-smells and SC-smells to discover their distributions and the degree of co-occurrence between them. Further, we want to compare their differences of co-occurrence degree between different project groups.

The occurrence of Dockerfile smells may depend on project’s characteristics, because different projects have different codes and goals. Our statistics showed that the distribution of Dockerfile smells among different projects very vary, some projects tend to have more Dockerfile smells than others (see Section IV-A, IV-B). Among individual projects, larger teams, popular, or owned by organizations may have a good work environment and be better prepared to write better Dockerfile. Also, projects with different programming languages tend to have different development paradigms [26]. Exploring whether and how project characteristics correlate with the Dockerfile smells occurrence can help us to find out the inherent factors that may affect the occurrence of Dockerfile smells in different projects. Thus, in our third question, we ask:

RQ3: Does smell occurrence depend on the characteristics of the project? In this question, we aim to investigate the relationship between project characteristics (e.g., programming language and popularity) and associated smell occurrence to find out whether the smell occurrence changes as the project characteristics differ.

Research and practice of software development are performed under various assumptions about Dockerfile configuration that have not been validated through extensive empirical studies. Therefore, several hypotheses can be made, that we will attempt to validate in our study:

H1. *It is difficult for developers to fully follow the Dockerfile best practices suggested by the Docker’s official documentation.*

H2. *Smells of Docker instructions and smells of Shell scripts co-exist in the Dockerfile codes.*

H3. Different project characteristics have impacts on Dockerfile smells occurrence.

III. EXPERIMENTAL SETUP

Figure 2 gives an overview of our study. Based on the research questions, we collect the Dockerfile data from thousands of selected GitHub projects, and perform quantitative studies on them.

A. DATA COLLECTION

1) CANDIDATE PROJECTS SELECTION

To select candidate projects, we start by retrieving the project basic information from GHTorrent dataset [27] (as of April 2018). GHTorrent is an effort to create a scalable, queriable, offline mirror of data offered through the Github REST API. It includes almost all historical data of projects (83M+), involving their issues, pull requests, commits, followers, etc. Note that, although the term “*project*” has been used to refer to a collection of interrelated repositories, we don’t distinguish between “*project*” and “*repository*” for the purpose of this study. To avoid biasing analysis, we remove projects that were forked from other projects, and we do not consider projects that have been deleted. Inspired by Cito et al.’s approach [10], we use Google’s BigQuery⁷ to select those projects that contain Dockerfile in their Git repositories and collect their Dockerfile contents.

Our initial dataset contains 16,959 projects, including their metadata (i.e., names, owner type, creation times, programming languages, number of stars, and number of contributors) and Dockerfile contents.

2) DATA PREPROCESSING

Further, we filter out programming languages that are not represented by many projects. From all selected projects, we calculate the number of projects per programming language and find that the Top-10 popular programming languages are Shell, Makefile, Ruby, PHP, Python, Java, HTML, CSS, JavaScript, and Go (the number of projects per programming language can be found in Table 3), which is consistent with the findings in Cito et al.’s study [10]. Then, we choose the projects that belong to the Top-10 popular programming languages for our quantitative study. The resulting study population for our analysis consists of 6,334 projects. In each project, there may exist many versions of Dockerfile in the development history. However, our study mainly focuses on the current occurrence of Dockerfile smells in GitHub projects and not its evolution, which is beyond the scope of this work. To be consistent with the version of the projects’ metadata (i.e., project characteristics), in this study we only consider the latest Dockerfile (most recent version, i.e., as of April 2018) of each project.

Table 1 presents aggregate descriptive statistics over the 6,334 projects, where “#” indicates the number of.

⁷https://bigquery.cloud.google.com/dataset/bigquery-public-data:github_repos

TABLE 1. Aggregate statistics of studied projects.

Statistic	Mean	St.Dev.	Min	Median	Max
#Dockerfile instructions	12.3	10.2	1.0	10.0	152.0
#project contributors	3.5	7.9	1.0	2.0	220.0
#project stars	75.6	526.5	0.0	1.0	17,899.0
GitHub age (in days)	1,016.0	354.3	291.0	991.0	3,494.0

On average, each project’s Dockerfile has 12.3 instructions (median: 10.0). The ages of our studied projects range from 291.0 days (~9.7 months) to 3,494.0 days (~9.6 years), the average age is 2.8 years (median: 2.7 years).

3) DOCKERFILE SMELLS DETECTION

By following the detection approach used in previous studies [10], [17], [28], in our study we mainly use the Haskell Dockerfile Linter⁸ to detect different Dockerfile smells. This tool parses the Dockerfile into an AST and performs rules on top of the AST. Also, it is standing on the shoulders of ShellCheck⁹ to lint the Bash code inside RUN instructions. Corresponding to the two types of smells we introduced in Section II-A.2, in Linter tool, rules with the prefix “DL” originate from hadolint and are intended to detect the DL-smells, e.g., “DL3007” indicates that “*Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag*”. Rules with the “SC” prefix originate from ShellCheck and are intended to detect the SC-smells, e.g., “SC1018” indicates that “*This is a unicode non-breaking space. Delete it and retype as space.*” We use the term “*total detected smells (by volume)*” to refer to all the smell instances detected in a Dockerfile.

In total, we detect 29,843 smells from those 6,334 projects, among them, 26,545 (88.9%) are DL-smells and 3,298 (11.1%) are SC-smells. The detailed discussion of Dockerfile smells coverage can be found in Section IV-A.

B. QUANTITATIVE ANALYSES

1) HYPOTHESIS TESTING

In our study, we use Chi-Square test [29] to see whether distributions of categorical variables differ from each other. Next, we use Wilcoxon test [30], which is a non-parametric test, to compare the difference between two distributions. If the significance level α is 0.05, p -value less than α means the test rejects the null hypothesis, which verifies that the two samples have different distributions at the significance level of 0.05. We also compute Cliff’s delta ($|\delta|$) [31], to measure the effect size that quantifies the difference between two distributions. Where the magnitude is assessed using the thresholds, i.e., $|\delta| < 0.147$ “negligible”, $|\delta| < 0.33$ “small”, $|\delta| < 0.474$ “medium”, otherwise “large”.

Further, to compare the differences between multiple distributions (> 2), we mainly use the Kruskal-Wallis test [32], which is an extension of the Wilcoxon test and can be used

⁸<https://hadolint.github.io/hadolint/>

⁹<https://github.com/koalaman/shellcheck>

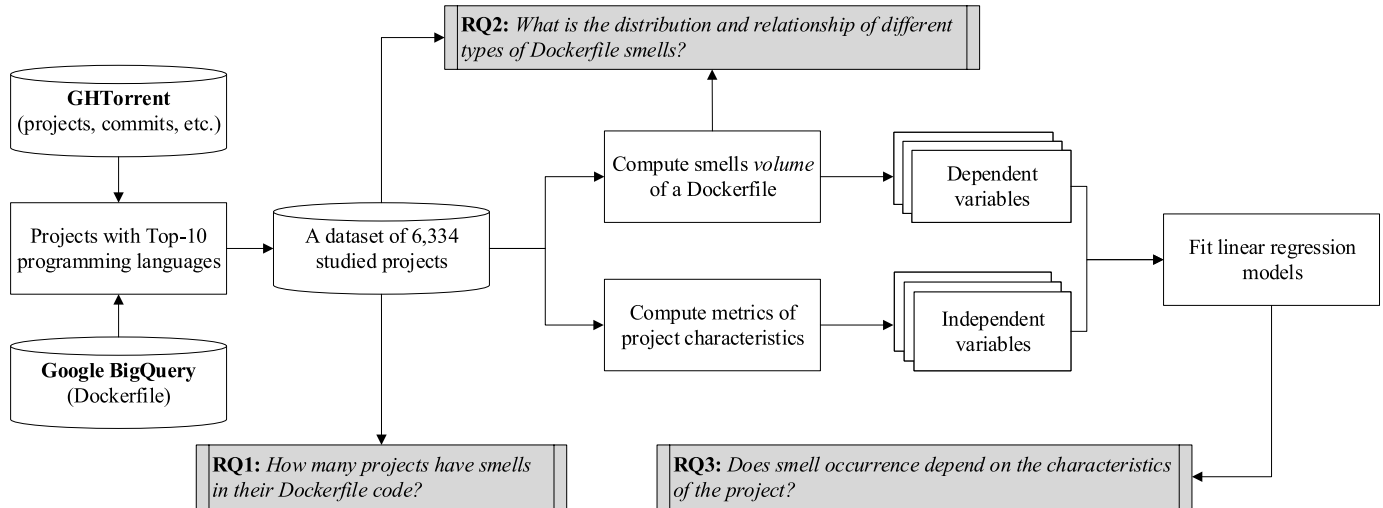


FIGURE 2. Overview of our study.

to test the hypothesis that a number of unpaired samples originate from the same population. Moreover, we use the Spearman rank correlation (ρ) [33] to discover the strength of a link between two sets of data. Spearman rank correlation is a non-parametric test that does not carry any assumptions about the distribution of the data. Specifically, $0.10 \leq \rho \leq 0.29$ represent a small correlation, $0.30 \leq \rho \leq 0.49$ represent a medium correlation, and $\rho \geq 0.50$ represent a large correlation.

2) REGRESSION MODELING

To statistically investigate whether smell occurrence depends on the project characteristics, we develop three linear regression models (via package `lm` in R), i.e., **Overall smells model**, **DL smells model**, and **SC smells model**. In our models, the variance inflation factors, which measure multicollinearity of the set of predictors, are safe (< 3). We use the adjusted R^2 statistic to evaluate the goodness-of-fit of our models. For each model variable, we report its coefficients, standard error, and significance level. We use red upward ($\nearrow$) and green downward ($\searrow$) arrows to indicate whether a variable has a direct or an inverse relationship, respectively, with the more smells. We consider the coefficients noteworthy if they are statistically significant at $p < 0.05$. Our models can be specified as follows:

$$\begin{aligned} \text{Type_Smells} \sim & nInstructions + nContributors \\ & + nStars + githubAge \\ & + \text{factor}(\text{ownerType}) \\ & + \text{factor}(\text{language}) \end{aligned} \quad (1)$$

The outcome (dependent) variables of the models are:

- **Overall_Smells**: the volume number of all smells;
- **DL_Smells**: the volume number of DL-smells;
- **SC_Smells**: the volume number of SC-smells;

Our independent variables of the models are mainly based on prior studies:

- **nInstructions**: number of instructions in a Dockerfile, as a proxy for the Dockerfile complexity [17], [20]. A large Dockerfile may have more smells than a small Dockerfile;
- **nContributors**: number of project contributors (submitted at least one commit), as a proxy for the project team size [34]. Large teams may be better prepared to write good Dockerfile;
- **nStars**: number of project stars, as proxy for the project popularity [35]. Popular projects may have different strategies for writing Dockerfile;
- **githubAge**: same as the approach proposed in the previous study [36], we measure the age of a project solely based its date of creation on GitHub. For each project, we count the number of days that have passed since it has been hosted on GitHub until April 2018. Smells occurrence may be different in old vs. young projects;
- **ownerType**: type of the project owner [37], i.e., “*Organization*” or “*User*” (as the baseline), as a proxy for the project internal property. *Organization* projects may have better development management than *User* projects.
- **language**: project’s programming language, one of the Top-10 programming languages as described in Data Collection. Projects with different programming languages are likely to have different development paradigms [26]. Effect coding [38] provides one way of using categorical predictor variables in the linear regression model. With effect coding, the experimental effect is analyzed as a set of contrasts that opposes all but one experimental condition to one given experimental condition. Also, the intercept is equal to the grand mean, and the slope for a contrast expresses the difference between a group and the grand mean [39]. Thus, we use effect coding to set the contrasts of this ten-way factor, i.e., comparing each level to the grand mean of all ten levels.

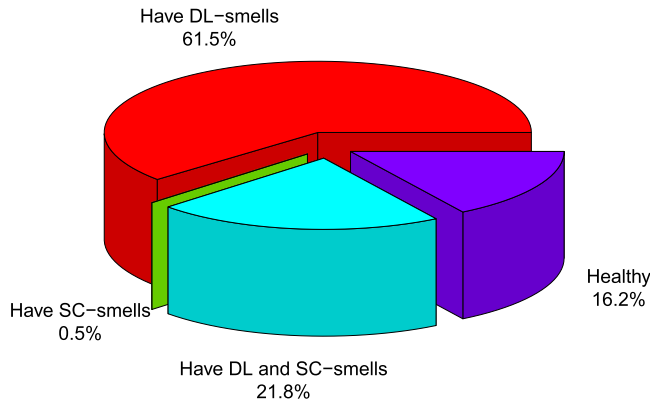


FIGURE 3. The distribution of projects w/o Dockerfile smells.

IV. EMPIRICAL STUDY RESULTS

Here, we discuss our empirical study results to answer our three research questions.

A. RQ1: HOW MANY PROJECTS HAVE SMELLS IN THEIR DOCKERFILE CODE?

1) BASIC SMELLS COVERAGE

Figure 3 presents the distribution of projects w/o Dockerfile smells, we find that 5,309 (83.8%) of 6,334 studied projects have at least one smell in their Dockerfile code, only 1,025 (16.2%) projects are “healthy” (without Dockerfile smells). This is consistent with the findings of a previous study [10], they found that only 16% of projects do not violate a single rule. Specifically, we find that 3,893 (61.5%) S-projects have DL-smells, 35 (0.5%) S-projects have SC-smells, while 1,381 (21.8%) S-projects have both DL-smells and SC-smells in their Dockerfile code. Thus, we find that, *Dockerfile smells are very common in GitHub projects, especially those smells that violate Dockerfile best practices, i.e., DL-smells.*

Further, we find that each S-project has an average of 5.6 smells (median: 4.0) in their Dockerfile code. The distribution of Dockerfile smells density (#smells/#instructions) is shown in Figure 4, on average, the smell density of S-projects is 0.62 (median: 0.43). It is noteworthy that, in 282 (5.3%) of 5,309 S-projects, all their instructions have smells (smells density=1). Thus, we find that *most of Dockerfile code (62%) have smells.*

Findings: Nearly 84% of GitHub projects in our dataset have smells in their Dockerfile code, especially the DL-smells. Specifically, an average of 62% of Dockerfile instructions are infected with smells. Thus, this is consistent with our hypothesis H1.

2) COVERAGE IN DIFFERENT PROJECTS

Next, we seek to investigate whether there are differences in smells coverage between different types of projects. Specifically, we compare the smells coverage between different project owner types, as shown in Table 2. We find that, although “User” owners have more projects than

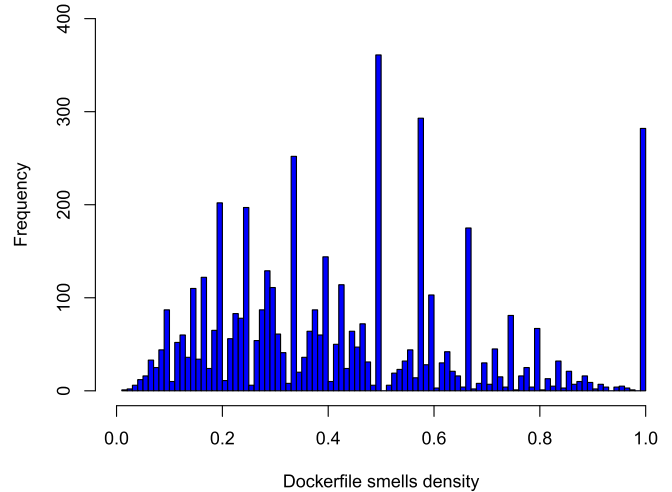


FIGURE 4. The distribution of Dockerfile smells density.

TABLE 2. Smells coverage in different project owner types.

Owner type	#projects	#S-projects	%S-projects	Avg. smells density
Organization	2,266	1,896	83.67%	61.42%
User	4,068	3,413	83.90%	61.58%

TABLE 3. Smells coverage in different programming languages.

Language	#projects↓	#S-projects	%S-projects	Avg. smells density
Shell	2,009	1,770	88.10%	62.63%
JavaScript	1,268	1,014	79.97%	56.81%
Python	973	816	83.86%	68.37%
Go	628	479	76.27%	70.98%
Java	364	305	83.79%	51.56%
Ruby	338	292	86.39%	59.72%
PHP	281	239	85.05%	51.18%
HTML	184	152	82.61%	60.13%
CSS	149	121	81.21%	61.98%
Makefile	140	121	86.43%	52.57%

“Organization” owners, there is no significant difference between their smells coverage (Chi-Square test: $\chi^2 = 0.04$, p -value=0.84). Nearly 84% projects have smells with about 61% instructions infected. Thus, we find that *Dockerfile smells coverage and density are not correlated with the project’s owner type.*

Also, we compare the smells coverage between different programming languages, as shown in Table 3. We find that projects with different programming languages significantly differ in smells coverage (Chi-Square test: $\chi^2 = 71.02$, p -value<0.001). Shell has the highest percentage of projects with smells (88.10%), followed by Makefile (86.43%) and Ruby (86.39%), while Go has the lowest (76.27%).

Findings: Regardless of the organization or the normal user, there is no significant difference in the smells coverage of their projects’ Dockerfiles. However, the smells coverage in projects with different programming languages may vary.

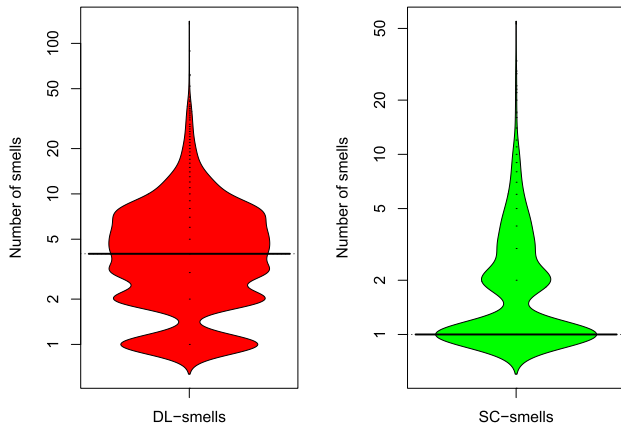


FIGURE 5. The distribution of DL-smells and SC-smells.

B. RQ2: WHAT IS THE DISTRIBUTION AND RELATIONSHIP OF DIFFERENT TYPES OF DOCKERFILE SMELLS?

1) DETAILED SMELLS DISTRIBUTION

We use the beanplots [40] to visualize the distributions of DL-smells and SC-smells per project in Figure 5 (Horizontal lines depict medians). In S-projects with DL-smells, each project has an average of 5.0 smells (median: 4.0). While in S-projects with SC-smells, each project has an average of 2.3 smells (median: 1.0). Thus, in S-projects, DL-smells appear more than SC-smells.

Further, we compare the different smells distribution between different types of projects. Figure 6 presents the distribution of smells in different project owner types (“Organization” and “User”). On average, each Organization project has 4.8 DL-smells (median: 4.0), while each User project has 5.1 DL-smells (median: 4.0). Using the Wilcoxon test, we have established that the difference between the two data sets is statistically significant ($p < 0.0001$), but their effect size is very small (Cliff’s delta: 0.06). Moreover, there is no significant difference in the distribution of SC-smells between Organization projects and User projects ($p = 0.92$). So we find that, the DL-smells distribution is weakly correlated with the project owner type, while the SC-smells distribution does not correlate with it.

Also, we compare the different smells distribution between different programming languages (as shown in Figure 7 and Figure 8). The median values in Shell, Ruby, Python, PHP, and JavaScript projects (4.0 DL-smells) are slightly higher than the median values (3.0 DL-smells) in CSS, Go, HTML, Java, and Makefile projects. The distribution of DL-smells per programming language is significantly different (Kruskal-Wallis test: $\chi^2 = 174.41$, $p < 0.001$). As for the SC-smells, we find that the median values in HTML, Java, Python, and Shell projects (2.0 DL-smells) are slightly higher than the median values (1.0 DL-smells) in CSS, Go, JavaScript, Makefile, PHP, and Ruby projects. The distribution of SC-smells per programming language is significantly different (Kruskal-Wallis test: $\chi^2 = 28.44$, $p < 0.001$).

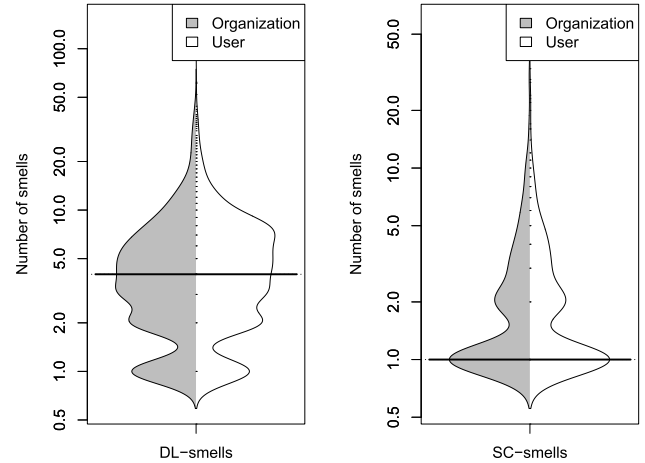


FIGURE 6. The distribution of smells in different project owner types.

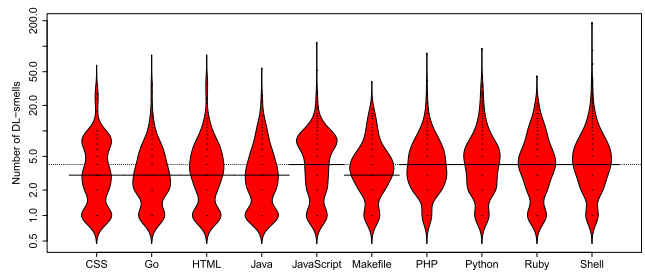


FIGURE 7. The distribution of DL-smells in different programming languages.

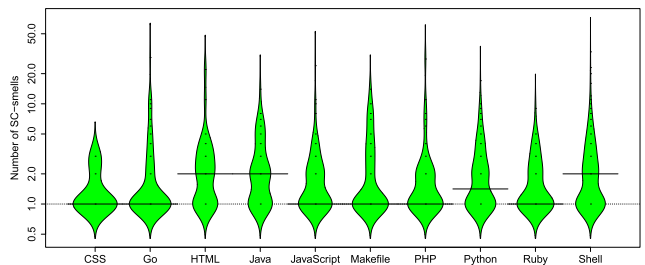


FIGURE 8. The distribution of SC-smells in different programming languages.

Comparing the two distributions, we find that programming languages are correlated with the smells distribution.

Findings: In general, DL-smells appear more than SC-smells in those S-projects. There is no obvious correlation between smells distribution and project owner type, but the smells distribution of projects with different programming languages may vary.

2) SMELLS CO-OCCURRENCE

To investigate the relationship between DL-smells and SC-smells, i.e., their co-occurrence, we compute the Spearman’s correlation coefficient between those two types of smells. Figure 9 presents a scatter graph showing the co-occurrence between DL-smells and SC-smells by volume. In overall S-projects, we find that DL-smells have positive

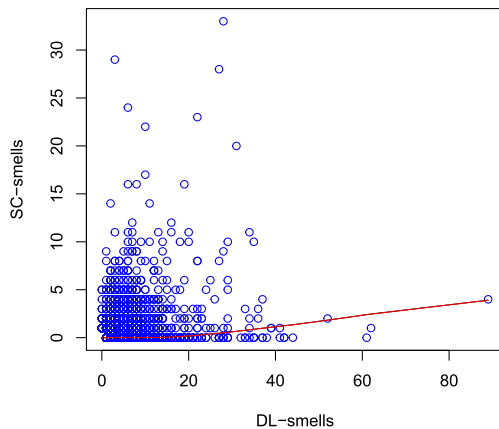


FIGURE 9. Co-occurrence between DL-smells and SC-smells (by volume).

TABLE 4. Correlation results in different project owner types.

Project owner type	Coef. (ρ)	Sign.
Organization	0.158	***
User	0.138	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

TABLE 5. Correlation results in different programming languages.

Project language	Coef. (ρ)	Sign.
Shell	0.114	***
JavaScript	0.003	
Python	0.255	***
Go	0.283	***
Java	0.133	*
Ruby	0.130	*
PHP	0.256	***
HTML	0.172	*
CSS	0.007	
Makefile	0.217	*

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

correlation with SC-smells ($\rho = 0.14$, $p < 0.001$), i.e., *high volume of DL-smells (or SC-smells) is a indication of the presence of high volume of SC-smells (or DL-smells) in a project.*

Further, we compare the difference of smells co-occurrence between different types of projects. Table 4 shows that, *both in Organization projects and User projects, there is a positive correlation between DL-smells and SC-smells with statistical significance.* However, the difference of correlation coefficient between the *Organization* projects and the *User* projects is very small ($\Delta = 0.02$).

Also, we compare the difference of smells co-occurrence between different programming languages, as shown in Table 5. We find that, *in most of the programming languages (8 out of 10), there are positive correlations between DL-smells and SC-smells with statistical significance.* Specifically, smells co-occurrence in Go ($\rho = 0.283$, $p < 0.001$), PHP ($\rho = 0.256$, $p < 0.001$), and Python ($\rho = 0.255$, $p < 0.001$) projects are higher than projects with other programming languages. While in JavaScript and CSS projects, there is no correlation between DL-smells and SC-smells.

Findings: A project that shows presence of large number of DL-smells (or SC-smells) moderately indicates presence of large number of SC-smells (or DL-smells). This is consistent with our hypothesis H2. However, the co-occurrence between DL-smells and SC-smells differs in programming languages.

C. RQ3: DOES SMELL OCCURRENCE DEPEND ON THE CHARACTERISTICS OF THE PROJECT?

1) OVERALL SMELLS MODEL

Table 6 gives the summary of the Overall smells model. The adjusted R^2 of the model is 24.5%. We find that, the factor *nInstructions* has a significant positive effect on the number of overall smells ($\rho=0.47$), holding all other variables constant. This is consistent with the notion that *more code tends to have more smells*. Also, project age (*githubAge*) has a significant positive effect ($\rho=0.11$), holding all other variables constant. So *old projects may have more smells than young projects*. We suspect that old projects may have more accumulated development tasks or code than young projects, which may induce more Dockerfile smells. We plan to investigate that in the future.

However, we find that factor *nStars* has a significant negative effect on the number of overall smells ($\rho = -0.03$), holding all other variables constant. This is consistent with the previous study [10], which found that Dockerfiles of the Top-1,000 most popular projects violate 3.5 rules on average while Dockerfiles of the Top-100 projects decrease to 3.2 rules. Thus, *popular projects are correlated with fewer smells*. Popular projects may have better strategies for writing Dockerfile than unpopular projects. Also, we find that there is no significant correlation between the overall smells number and the project owner type, which is consistent with our findings reported before.

We find that there are some significant correlations between the number of overall smells and certain programming languages. Compared to the overall mean of ten programming languages, projects using JavaScript and Python have significant positive effects on the number of overall smells ($\rho=0.05$ and $\rho=0.09$), while Java has a significant

TABLE 6. Results of Overall smells model.

	Coef.	Err.	$Pr(< z)$	Sign.	Relationship
(Intercept)	-0.02	0.02	0.259		—
nInstructions	0.47	0.01	0.000	***	↗
nContributors	-0.02	0.01	0.053		—
nStars	-0.03	0.01	0.020	*	↘
githubAge	0.11	0.01	0.000	***	↗
ownerType=Organization	-0.03	0.02	0.170		—
language=CSS	-0.04	0.07	0.567		—
language=Go	-0.07	0.04	0.060		—
language=HTML	0.01	0.06	0.871		—
language=Java	-0.11	0.04	0.012	*	↘
language=JavaScript	0.05	0.03	0.049	*	↗
language=Makefile	0.02	0.07	0.807		—
language=PHP	-0.05	0.05	0.342		—
language=Python	0.09	0.03	0.001	**	↗
language=Ruby	0.02	0.05	0.644		—
language=Shell	0.07	0.02	0.170		—

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

negative effect ($\rho = -0.11$), holding all other variables constant.

Findings: Popular and young projects tend to have fewer Dockerfile smells. Also, projects using Java are likely to have fewer Dockerfile smells. However, more Dockerfile smells can be found in those JavaScript projects or Python projects. We find no evidence that project's contributor team size and owner type have effects on the Dockerfile overall smells occurrence. Not all project characteristics correlate with the occurrence of Dockerfile smells, we therefore note that hypothesis H3 is disputable.

2) DL SMELLS MODEL

Table 7 gives the summary of the DL smells model. The adjusted R^2 of the model is 20.0%. Different from the results of Overall smells model, the factor *nContributors* has a significant negative effect on the number of DL-smells ($\rho = -0.03$), holding all other variables constant. Thus, projects with large contributor teams tend to have fewer DL-smells than small teams. More contributors may be better prepared to write better Dockerfile.

Interestingly, we find that the *Organization* projects (*ownerType=Organization*) have a significant negative effect ($\rho = -0.06$), holding all other variables constant. Thus, *Organization* projects tend to have fewer DL-smells than *User* projects. *Organization* projects may have higher standards and more inspection methods than *User* projects when writing Dockerfiles, and they are more likely to follow the public Dockerfile best practices, so their Dockerfiles are less likely to have DL-smells.

We find that there are some significant correlations between the number of DL-smells and certain programming languages. Compared to the overall mean of ten programming languages, projects using JavaScript and Python have significant positive effects on the number of DL-smells ($\rho=0.11$ and $\rho=0.12$), while projects using Java and Go have significant negative effects ($\rho = -0.13$ and $\rho = -0.08$), holding all other variables constant.

TABLE 7. Results of DL smells model.

	Coef.	Err.	$Pr(< z)$	Sign.	Relationship
(Intercept)	-0.02	0.02	0.356	—	—
nInstructions	0.42	0.01	0.000	***	↗
nContributors	-0.03	0.01	0.023	*	↘
nStars	-0.03	0.01	0.006	**	↘
githubAge	0.12	0.01	0.000	***	↗
ownerType=Organization	-0.06	0.02	0.018	*	↘
language=CSS	0.01	0.07	0.894	—	—
language=Go	-0.08	0.04	0.022	*	↘
language=HTML	-0.01	0.06	0.824	—	—
language=Java	-0.13	0.05	0.004	**	↘
language=JavaScript	0.11	0.03	0.000	***	↗
language=Makefile	-0.06	0.07	0.399	—	—
language=PHP	-0.04	0.05	0.442	—	—
language=Python	0.12	0.03	0.000	***	↗
language=Ruby	0.04	0.05	0.340	—	—
language=Shell	0.05	0.02	0.056	—	—

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

TABLE 8. Results of SC smells model.

	Coef.	Err.	$Pr(< z)$	Sign.	Relationship
(Intercept)	-0.02	0.02	0.366	—	—
nInstructions	0.34	0.01	0.000	***	↗
nContributors	0.01	0.01	0.664	—	—
nStars	0.01	0.01	0.586	—	—
githubAge	0.02	0.01	0.121	—	—
ownerType=Organization	0.06	0.03	0.012	*	↗
language=CSS	-0.16	0.07	0.027	*	↘
language=Go	0.03	0.04	0.504	—	—
language=HTML	0.07	0.06	0.243	—	—
language=Java	0.02	0.05	0.684	—	—
language=JavaScript	-0.15	0.03	0.000	***	↘
language=Makefile	0.23	0.07	0.001	**	↗
language=PHP	-0.04	0.05	0.435	—	—
language=Python	-0.03	0.03	0.320	—	—
language=Ruby	-0.06	0.05	0.190	—	—
language=Shell	0.08	0.03	0.001	**	↗

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

Findings: Projects owned by organizations or have large contributors team are more likely to have fewer DL-smells. Also, projects with Go and Java are correlated with fewer DL-smells. But projects with JavaScript and Python are correlated with more DL-smells.

3) SC SMELLS MODEL

Table 8 gives the summary of the SC smells model. The adjusted R^2 of the model is 13.2%. We find that, the *Organization* projects (*ownerType = Organization*) have a significant positive effect on the number of SC-smells ($\rho=0.06$), holding all other variables constant. Thus, *Organization* projects tend to have more SC-smells than *User* projects. This is in contrast to our findings in the DL smells model. We assume that *Organization* projects may need to handle more shell scripts than *User* projects, which requires further verification in the future. Different from the results of DL smells model, we find no evidence that SC-smells occurrence depends on the project's contributors team size (*nContributors*), popularity (*nStars*), and age (*githubAge*).

However, we find that there are some significant correlations between the number of SC-smells and certain programming languages. Compared to the overall mean of ten programming languages, projects using Makefile and Shell have significant positive effects on the number of SC-smells ($\rho=0.23$ and $\rho=0.08$), while projects using CSS and JavaScript have significant negative effects ($\rho = -0.16$ and $\rho = -0.15$), holding all other variables constant.

Findings: Projects owned by regular users tend to have fewer SC-smells. Also, projects using CSS or JavaScript are more likely to have fewer SC-smells. However, Makefile or Shell projects are correlated with more SC-smells. Moreover, project's contributors team size, popularity, and age, are not correlated with the occurrence of SC-smells.

V. DISCUSSION

A. QUALITATIVE ANALYSIS

In our quantitative study, we find that Dockerfile smells occurrence differs in different programming languages. Among them, projects using Shell, Makefile, and Ruby tend

to have more smells than other projects (Section IV-A). To gain an initial understanding of why Shell, Makefile, and Ruby projects tend to be infected with Dockerfile smells, we randomly select 60 S-projects (20 for each programming language) from our dataset and manually analyze their Dockerfile contents. We find that, Shell projects tend to use the “*apt-get update*” and “*apt-get install*” in the RUN instruction, which is prone to go against the best practice: “*Delete the apt-get lists after installing something*”. E.g., in the *ch3ts/named* project, as the best practice suggested, developers should clean up the *apt-get* cache and remove */var/lib/apt/lists* after updating and installing packages *wget*, *openssl*, etc. Otherwise it will cause smell.

Smell related to *apt-get update & install*:

```
RUN apt-get update && apt-get -y install wget openssl ...
+ && apt-get clean \
+ && rm -rf /var/lib/apt/lists/*
```

In Makefile projects, a very common smell is that developers use the “*cd*” command in the RUN instruction to switch to a new directory. However, most commands can work with absolute paths and it is in most cases not necessary to change directories. E.g., in *hiteshjasani/dock* project, developers should use the WORKDIR instruction to change the current working directory.

Smell related to *cd*:

```
- RUN cd /usr/local/src && \
    git clone https://github.com/nimrod-code/nimble.git
+ WORKDIR /usr/local/src
+ RUN git clone https://github.com/nimrod-code/nimble.git
```

As for Ruby projects, we find a very common smell that developers usually use the ADD instruction, especially when it does not require its tar auto-extraction capability for files and folders, which will cause the smell. E.g., in *kunday/cloudformer* project, developers should use COPY instead of ADD for files and folders.

Smell related to *add & copy*:

```
- ADD . /usr/local/app
+ COPY . /usr/local/app
```

Interestingly, we find 17 out of 60 projects still use the MAINTAINER instruction, which is deprecated since Docker 1.13.0¹⁰. E.g., in the *rwylabs/consul* project, developers should remove the maintainer information “*Leon de Jager <ldejager@rwylabs.io>*”, otherwise it will cause smell. It also reminds developers that they should keep themselves updated with the changing technology.

Smell related to *maintainer*:

```
FROM rwylabs/consul:latest
- MAINTAINER Leon de Jager <ldejager@rwylabs.io>
```

B. PRACTICAL IMPLICATIONS

From our quantitative study results, we can distill some implications for researchers, developers, and tool builders.

1) FOR RESEARCHERS

Relationship between instructions and smells. Our quantitative analysis finds that 62% of Dockerfile instructions are infected with smells, especially the DL-smells (Section IV-A and IV-B). Researchers should further explore in the Dockerfile configuration, whether some instructions are more likely to stick to smells than other instructions.

Configuration patterns in different programming languages. Our statistics show that the project’s programming language is correlated with Dockerfile smells occurrence, including its coverage and distribution (Section IV-A, IV-B, and IV-C). Whether developers tend to have different Dockerfile configuration patterns in different programming languages, researchers can use some association rule mining or clustering methods to explore it further.

Smells occurrence in other dimensions. Moreover, how the influence of Dockerfile smells occurrence differs in other dimensions should be further empirically evaluated. E.g., researchers should further investigate whether Dockerfile smells occurrence depends on the project’s release frequency, code productivity, or bug-introducing changes. With much more data and careful project classification along different dimensions, some patterns may become apparent.

Investigation of smell-introducing reasons. Our study shows that it is difficult for developers to fully follow the official Dockerfile best practices. Researchers should further explore the reasons for the smell-introducing of Dockerfile, which needs more qualitative analyses, e.g., manual analysis, survey, and interview. Our findings motivate the need for collecting more empirical evidence that helps developers, who seek to better write Dockerfile in their projects without arbitrary decisions.

2) FOR DEVELOPERS

Large team is helpful. Our quantitative study results show that more contributors are correlated with fewer DL-smells (Section IV-C). A direct implication is that, if a project has a large developer community, then the probability of infecting DL-smells might be reduced, as more developers have more eyes to check the Dockerfile quality. Therefore, the issue of how to manage and help a group of developers configure the best Dockerfile needs to be addressed.

More attention is needed. In current practices, smells are very common in Dockerfile codes (Section IV-A). Developers should seriously pay more attention to the Dockerfile quality when writing Dockerfile. A natural suggestion is that developers should follow the official Dockerfile best practices as much as possible. Moreover, focused training sessions are also crucial to increase awareness of Dockerfile quality among developers.

Quality detection is important. During the Docker image building, developers should use the Dockerfile quality auto-detection tool (e.g., the Haskell Dockerfile Linter) to iteratively detect smells as much as possible before actually deploying the Dockerfile.

¹⁰<https://github.com/lukasmartinelli/hadolint/issues/71>

Improving Dockerfile content is beneficial. In addition, our studies reveal that project's characteristics (e.g., programming language) and Dockerfile complexity (i.e., number of instructions) have important effects on the Dockerfile smells occurrence (Section IV-A, IV-B, and IV-C). Developers should select appropriate instructions and simplify their Dockerfile content and optimize the image structures, i.e., reducing image layers and changing instruction orders.

3) FOR TOOL BUILDERS

Sophisticated detection tools. Our statistical results indicate that smells extensively exist in today's Dockerfile codes (Section IV-A). Hence, tool builders may look into creating sophisticated detection tools that enhance Dockerfile quality and integrate with different IDEs, native or extended (via plug-ins). Those modern tools should raise an alarm (e.g., in the form of warnings), to attract developers' attention towards potential smells in the Dockerfile configuration, early on and rectify them.

Intelligent management tools. Based on a collection of common practices followed globally or within an organization, tool builders may develop intelligent management tools to ensure the consistency and effectiveness of the Dockerfile environment, e.g., recommending appropriate base image or instructions, predicting Docker build failure or latency. Across the industry, those intelligent management tools would prohibit some of the Dockerfile smells.

C. THREATS TO VALIDITY

Here we discuss some threats to validity that we have identified in the course of this study.

1) INTERNAL VALIDITY

In our study, we rely on the data collected mostly from GHTorrent and the GitHub public data that we query from Google BigQuery. However, those two datasets still exist some mistakenly stored values. This may lead to some bias in our study. Nonetheless, we have manually checked and found that phenomenon is not common. For multi-language projects, we consider only the dominant language, i.e., the language with the highest number of lines of code in a project. Moreover, we only look at the latest version of Dockerfile (as of April 2018) of each project, which may introduce some threats, although we do not find evidence for it. In future work, we plan to investigate how Dockerfile smells evolve among different Dockerfile versions.

In the regression modeling, we use five project characteristics as independent variables: *owner type*, *project age*, *contributors size*, *number of stars*, and *programming language*. We are aware that these factors are not fully comprehensive and using other factors may affect our results. E.g., those projects may have many different application domains (e.g., system software, web frameworks, and non-web libraries), which may cause some bias, although in our manual examination we do not find evidence for it. In future work, we plan to study how the occurrence of Dockerfile smells varies in

different application domains. We note that our models' fit to the data is around 20% of the deviance, and lower for the SC smells model. That is not necessarily a problem for our purposes as we are only interested in the coefficients' effect and not relying on the models to explain the full phenomena of Dockerfile smells occurrence, which would require many more variables, and is beyond the scope of this work.

2) EXTERNAL VALIDITY

Although our dataset consists of over 6,000 projects, the results may not represent all real-world projects. Also, we only consider open-source projects that are hosted on GitHub. Thus, our findings cannot be assured to generalize to private personal projects or projects hosted on other services, e.g., Bitbucket or GitLab, although there is no inherent reason why they would be biased. In particular, the occurrence of Dockerfile smells might be different in closed-source business environments. In future work, we plan to explore the difference in Dockerfile smells occurrence between open-source projects and industrial projects.

VI. CONCLUSION AND FUTURE WORK

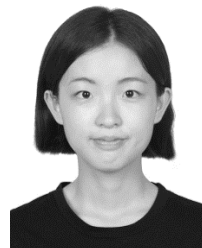
Due to limitations of experience and capabilities, developer's Dockerfile code may violate the recommended best practices, thereby causing Dockerfile smells. To help developers gain some insights into the occurrence of Dockerfile smells in open-source software, this paper presents an empirical study of the Dockerfile smells occurrence in a large-scale dataset of GitHub projects, including its coverage, distribution, and co-occurrence, as well as the correlation with the project characteristics. Some of our findings show promise for further investigation and may lead to enhanced Dockerfile best practices, initially tailored to the GitHub OSS community.

As future work, we plan to examine the detailed differences and patterns in the contents of Dockerfile under different project characteristics. We also plan to investigate the differences in the occurrence of Dockerfile smells between open-source projects and industrial projects.

REFERENCES

- [1] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.
- [2] C. Anderson, "Docker [software engineering]," *IEEE Softw.*, vol. 32, no. 3, p. 102–c3, Apr. 2015.
- [3] C. Boettiger, "An introduction to Docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015.
- [4] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 497–498.
- [5] T. Sharma and D. Spinellis, "A survey on software smells," *J. Syst. Softw.*, vol. 138, pp. 158–173, Apr. 2018.
- [6] T. Sharma, M. Frangkoulis, S. Rizou, M. Bruntink, and D. Spinellis, "Smelly relations: Measuring and understanding database schema quality," in *Proc. 40th Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, 2018, pp. 55–64.
- [7] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in Java," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 459–469.
- [8] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (Mis)use travis CI," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 33–50, Jan. 2020.

- [9] A. Mouat, *Using Docker: Developing and Deploying Software with Containers*. Sebastopol, CA, USA: O'Reilly Media, 2015.
- [10] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 323–333.
- [11] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code—an empirical study," in *Proc. IEEE/ACM 12th Working Conf. Mining Softw. Repositories*, May 2015, pp. 45–55.
- [12] T. Sharma, M. Frangkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proc. 13th Int. Workshop Mining Softw. Repositories (MSR)*, 2016, pp. 189–200.
- [13] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation (Adobe Reader)*. London, U.K.: Pearson, 2010.
- [14] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 307–319.
- [15] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How do system administrators resolve access-denied issues in the real world?" in *Proc. CHI Conf. Hum. Factors Comput. Syst. (CHI)*, 2017, pp. 348–361.
- [16] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 105–115.
- [17] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, "One size does not fit all: An empirical study of containerized continuous deployment workflows," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2018, pp. 295–306.
- [18] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: Recommending updates of Dockerfiles via software environment analysis," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 796–801.
- [19] Y. Zhang, H. Wang, and V. Filkov, "A clustering-based approach for mining dockerfile evolutionary trajectories," *Sci. China Inf. Sci.*, vol. 62, no. 1, Jan. 2019.
- [20] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "An insight into the impact of Dockerfile evolutionary trajectories on quality and latency," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2018, pp. 138–143.
- [21] K. Yin, W. Chen, J. Zhou, G. Wu, and J. Wei, "STAR: A specialized tagging approach for docker repositories," in *Proc. 25th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2018, pp. 426–435.
- [22] J. Zhou, W. Chen, G. Wu, and J. Wei, "Semitagrec: A semi-supervised learning based tag recommendation approach for Docker repositories," in *Proc. Int. Conf. Softw. Syst. Reuse (ICSR)*. Cham, Switzerland: Springer, 2019, pp. 132–148.
- [23] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in Dockerfiles," *IEEE Access*, vol. 7, pp. 63650–63659, 2019.
- [24] J. Xu, Y. Wu, Z. Lu, and T. Wang, "Dockerfile TF smell detection based on dynamic and static analysis methods," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2019, pp. 185–190.
- [25] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, A System of Patterns*. vol. 1. Hoboken, NJ, USA: Wiley, 1996.
- [26] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf.*, Jul. 2013, pp. 303–312.
- [27] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a fire-hose," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, Jun. 2012, pp. 12–21.
- [28] G. Schermann, S. Zumberi, and J. Cito, "Structured information on state and evolution of dockerfiles on github," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, 2018, pp. 26–29.
- [29] A. Satorra and P. M. Bentler, "A scaled difference chi-square test statistic for moment structure analysis," *Psychometrika*, vol. 66, no. 4, pp. 507–514, Dec. 2001.
- [30] E. A. Gehan, "A generalized Wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, no. 1/2, p. 203, Jun. 1965.
- [31] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations," *Univ. Psychol.*, vol. 10, no. 2, pp. 545–555, 2011.
- [32] N. Breslow, "A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship," *Biometrika*, vol. 57, no. 3, pp. 579–594, 1970.
- [33] J. H. Zar, "Significance testing of the spearman rank correlation coefficient," *J. Amer. Stat. Assoc.*, vol. 67, no. 339, pp. 578–580, Sep. 1972.
- [34] Y. Zhang, Y. Yu, H. Wang, B. Vasilescu, and V. Filkov, "Within-ecosystem issue linking: A large-scale study of rails," in *Proc. 7th Int. Workshop Softw. Mining*, 2018, pp. 12–19.
- [35] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 334–344.
- [36] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. L. Traon, "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 188–197.
- [37] J. Jiang, D. Lo, X. Ma, F. Feng, and L. Zhang, "Understanding inactive yet available assignees in GitHub," *Inf. Softw. Technol.*, vol. 91, pp. 44–55, Nov. 2017.
- [38] H. Alkharusi, "Categorical variables in regression analysis: A comparison of dummy and effect coding," *Int. J. Educ.*, vol. 4, no. 2, p. 202, Jun. 2012.
- [39] N. J. Salkind, *Encyclopedia of Research Design*, vol. 1. Newbury Park, CA, USA: Sage, 2010.
- [40] P. Kampstra, "Beanplot: A boxplot alternative for visual comparison of distributions," *J. Stat. Softw.*, vol. 28, no. 1, pp. 1–9, 2008.



YIWEN WU received the B.S. degree in network engineering from the National University of Defense Technology (NUDT), China, in 2018, where she is currently pursuing the M.S. degree in software engineering with the PDL Laboratory. Her research interests include computer society and empirical software engineering.



YANG ZHANG received the Ph.D. degree in software engineering from the National University of Defense Technology (NUDT), China, in 2019. He is currently an Assistant Professor with the PDL Laboratory, NUDT. His research interests include empirical software engineering, social network analysis, and DevOps.



TAO WANG received the Ph.D. degree in computer science from the National University of Defense Technology (NUDT), China, in 2014. He is currently an Assistant Professor with the PDL Laboratory, NUDT. His research interests include open-source software engineering, machine learning, data mining, and knowledge discovering in open-source software.



HUAIMIN WANG received the Ph.D. degree in computer science from the National University of Defense Technology (NUDT), China, in 1992. He is currently a Professor and the Chief of the Department of Educational Affairs, NUDT. His current research interests include middleware, software agent, and trustworthy computing. He was a recipient of the Chang Jiang Scholars Program Professor and the Distinct Young Scholar.

...